# Embedded SQL programming

DB2 Information Management Software

**http://www-136.ibm.com/developerworks/db2**

## Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

# Section 1. Before you start

## What is this tutorial about?

This tutorial introduces you to embedded SQL programming and walks you through the basic steps used to construct an embedded SQL application. This tutorial also introduces you to the process used to convert one or more high-level programming language source code files containing embedded SQL into an executable application. In this tutorial, you will learn:

- How SQL statements are embedded in a high-level programming language source code file
- The steps involved in developing an embedded SQL application
- What host variables are, how they are created, and how they are used
- What indicator variables are, how they are created, and when they are used
- How to analyze the contents of an SQLCA data structure variable
- How to establish a database connection from an embedded SQL application
- How to capture and process errors when they occur
- How to convert source code files containing embedded SQL into an executable application

This is the third in a series of seven tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The material in this tutorial primarily covers the objectives in Section 3 of the exam, entitled "Embedded SQL programming." You can view these objectives at: *http://www.ibm.com/certify/tests/obj703.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Edition for reference.

---

## Who should take this tutorial?

To take the DB2 V8 Family Development exam, you must have already passed the DB2 UDB V8.1 Family Fundamentals exam (Exam 700). You can use the DB2 Family Fundamentals tutorial series (see Resources on page 34) to prepare for that test. It is a very popular tutorial series that has helped many people understand the fundamentals of the DB2 family of products.

Although not all materials discussed in the Family Fundamentals tutorial series are required to understand the concepts described in this tutorial, you should have a basic knowledge of:

- DB2 instances

- Databases
- Database objects
- DB2 security

This tutorial is one of the tools that can help you prepare for Exam 703. You should also take advantage of the Resources on page 34 identified at the end of this tutorial for more information.

---

## About the author

Roger E. Sanders is a database performance engineer with Network Appliance, Inc. He has been designing and developing database applications for more than 18 years and he is the author of eight books on DB2 Universal Database, including *DB2 Universal Database v8.1 Certification Exam 703 Study Guide*, *DB2 Universal Database v8.1 Certification Exams 701 and 706 Study Guide*, and *DB2 Universal Database v8.1 Certification Exam 700 Study Guide*. In addition, Roger is a regular contributor to *DB2 Magazine* and he frequently presents at International DB2 User's Group (IDUG) and regional DB2 User's Group (RUG) conferences. Roger holds eight IBM DB2 certifications, including:

- IBM Certified Advanced Database Administrator -- DB2 Universal Database V8.1 for Linux, UNIX, and Windows
- IBM Certified Database Administrator -- DB2 Universal Database V8.1 for Linux, UNIX, and Windows
- IBM Certified Application Developer -- DB2 Universal Database V8.1 Family
- IBM Certified Database Associate -- DB2 Universal Database V8.1 Family.
- IBM Certified Advanced Technical Expert -- DB2 for Clusters

You can reach Roger at *rsanders@netapp.com*.

---

## Notices and trademarks

# Section 2. Introduction to embedded SQL programming

## Structured Query Language and embedded SQL

*Structured Query Language* (SQL) is a standardized language used to manipulate database objects and the data they contain. SQL is comprised of several different statements that are used to define, alter, and destroy database objects, as well as add, update, delete, and retrieve data values. However, SQL is nonprocedural, and therefore is not a general-purpose programming language. (SQL statements are executed by the DB2 Database Manager, not by the operating system.) As a result, database applications are usually developed by combining the decision and sequence control of a high-level programming language with the data storage, manipulation, and retrieval capabilities of SQL. Several methods are available for merging SQL with a high-level programming language, but the simplest approach is to embed SQL statements directly into the source code file(s) that will be used to create an application. This technique is referred to as *embedded SQL programming*.

One of the drawbacks to developing applications using embedded SQL is that high-level programming language compilers do not recognize, and therefore cannot interpret, SQL statements encountered in a source code file. Because of this, source code files containing embedded SQL statements must be preprocessed (using a process known as *precompiling*) before they can be compiled (and linked) to produce a database application. To facilitate this preprocessing, each SQL statement coded in a high-level programming language source code file must be prefixed with the keywords `EXEC SQL` and terminated with either a semicolon (in C/C++ or FORTRAN) or the keywords `END_EXEC` (in COBOL). When the preprocessor (a special tool known as the *SQL precompiler*) encounters these keywords, it replaces the text that follows (until a semicolon (`;`) or the keywords `END-EXEC` is found) with a DB2 UDB-specific function call that forwards the specified SQL statement to the DB2 Database Manager for processing.

Likewise, the DB2 Database Manager cannot work directly with high-level programming language variables. Instead, it must use special variables known as *host variables* to move data between an application and a database. (We will take a closer look at host variables in Declaring host variables on page 8.) Host variables look like any other high-level programming language variable; to be set apart, they must be defined within a special section known as a *declare section*. Also, in order for the SQL precompiler to distinguish host variables from other text in an SQL statement, all references to host variables must be preceded by a colon (`:`).

---

## Static SQL

A *static SQL* statement is an SQL statement that can be hardcoded in an application program at development time because information about its

structure and the objects (i.e., tables, column, and data types) it is intended to interact with is known in advance. Since the details of a static SQL statement are known at development time, the work of analyzing the statement and selecting the optimum data access plan to use to execute the statement is performed by the DB2 optimizer as part of the development process. Because their operational form is stored in the database (as a *package*) and does not have to be generated at application run time, static SQL statements execute quickly.

The downside to this approach is that all static SQL statements must be prepared (in other words, their access plans must be generated and stored in the database) before they can be executed. Furthermore, static SQL statements cannot be altered at run time, and each application that uses static SQL must *bind* its operational package(s) to every database with which the application will interact. Additionally, because static SQL applications require prior knowledge of database objects, changes made to those objects after an application has been developed can produce undesirable results.

The following are examples of static SQL statements:

```
SELECT COUNT(*) FROM EMPLOYEE

UPDATE EMPLOYEE SET LASTNAME = 'Jones' WHERE EMPID = '001'

SELECT MAX(SALARY), MIN(SALARY) INTO :MaxSalary, :MinSalary FROM EMPLOYEE
```

Generally, static SQL statements are well suited for high-performance applications that execute predefined operations against a known set of database objects.

---

# Dynamic SQL

Although static SQL statements are relatively easy to incorporate into an application, their use is somewhat limited because their format must be known in advance. *Dynamic SQL* statements, on the other hand, are much more flexible because they can be constructed at application run time; information about a dynamic SQL statement's structure and the objects with which it plans to interact does not have to be known in advance. Furthermore, because dynamic SQL statements do not have a precoded, fixed format, the data object(s) they reference can change each time the statement is executed.

Even though dynamic SQL statements are generally more flexible than static SQL statements, they are usually more complicated to incorporate into an application. And because the work of analyzing the statement to select the best data access plan is performed at application run time (again, by the DB2 optimizer), dynamic SQL statements can take longer to execute than their static SQL counterparts. (Since dynamic SQL statements can take advantage of the database statistics available at application run time, there are some cases in

which a dynamic SQL statement will execute faster than an equivalent static SQL statement, but those are the exception and not the norm.)

The following are examples of dynamic SQL statements:

```
SELECT COUNT(*) FROM ?

INSERT INTO EMPLOYEES VALUES (?, ?)

DELETE FROM DEPARTMENT WHERE DEPTID = ?
```

Generally, dynamic SQL statements are well suited for applications that interact with a rapidly changing database or that allow users to define and execute ad-hoc queries.

# Section 3. Constructing an embedded SQL application

## Declaring host variables

Earlier, we saw that the DB2 Database Manager relies on *host variables* to move data between an application and a database. We also saw that we distinguish host variables from other high-level programming language variables by defining host variables in a special section known as a *declare section*. So just how do we write declare sections?

The beginning of a declare section is defined by the `BEGIN DECLARE SECTION` SQL statement, while the end is defined by the `END DECLARE SECTION` statement. Thus, a typical declare section in a C/C++ source code file would look something like:

```
EXEC SQL BEGIN DECLARE SECTION
    char    EmployeeID[7];
    double  Salary;
EXEC SQL END DECLARE SECTION
```

A declare section can be coded anywhere high-level programming language variable declarations can be coded in a source code file. Although a source code file typically contains only one declare section, multiple declare sections are allowed.

Host variables that transfer data to a database are known as *input host variables*, while host variables that receive data from a database are known as *output host variables*. Regardless of whether a host variable is used for input or output, its attributes must be appropriate for the context in which it is used. Therefore, you must define host variables in such a way that their data types and lengths are compatible with the data types and lengths of the columns they are intended to work with: When deciding on the appropriate data type to assign to a host variable, you should obtain information about the column or special register that the variable will be associated with and refer to the conversion charts found in the *IBM DB2 Universal Database Application Development Guide: Programming Client Applications* documentation (see Resources on page 34). Also, keep in mind that each host variable used in an application must be assigned a unique name. Duplicate names in the same file are not allowed, even when the host variables are defined in different declare sections. A tool known as the *Declaration Generator* can be used to generate host variable declarations for the columns of a given table in a database. This tool creates embedded SQL declaration source code files, which can easily be inserted into C/C++, Java language, COBOL, and FORTRAN applications. For more information about this utility, refer to the `db2dclgen` command in the *DB2 UDB Command Reference* product documentation.

How is a host variable used to move data between an application and a database? The easiest way to answer this question is by examining a simple embedded SQL source code fragment. The following C source code fragment

illustrates the proper use of host variables:

```
...
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    char     EmployeeNo[7];
    char     LastName[16];
EXEC SQL END DECLARE SECTION;
...

// Retrieve A Record From The Database
EXEC SQL SELECT EMPNO, LASTNAME
    INTO :EmployeeNo, :LastName
    FROM EMPLOYEE
    WHERE EMPNO = '000100';

// Do Something With The Results
...
```

---

# Declaring indicator variables

By default, columns in a DB2 UDB database table can contain null values. Because null values are not stored the same way conventional data is stored, special provisions must be made if an application intends to work with null data. Null values cannot be retrieved and copied to host variables in the same manner that other data values can. Instead, a special flag must be examined to determine whether a specific value is meant to be null. And in order to obtain the value of this flag, a special variable known as an *indicator variable* (or *null indicator variable*) must be associated with the host variable that has been assigned to a nullable column.

Because indicator variables must be accessible by both the DB2 Database Manager and the application program, they must be defined inside a declare section and they must be assigned a data type that is compatible with the DB2 UDB `SMALLINT` data type. Thus, the code used to define an indicator variable in a C/C++ source code file will look something like:

```
EXEC SQL BEGIN DECLARE SECTION
    short   SalaryNullIndicator;
EXEC SQL END DECLARE SECTION
```

An indicator variable is *associated* with a specific host variable when it follows the host variable in an SQL statement. Once an indicator variable has been associated with a host variable, it can be examined as soon as its corresponding host variable has been populated. If the indicator variable contains a negative value, a null value was found and the value of the corresponding host variable should be ignored. Otherwise, the value of the corresponding host variable is valid.

Again, in order to understand how indicator variables are used, it helps to look at an example embedded SQL source code fragment. The following code, written in the C programming language, shows one example of how indicator variables are defined and used:

```
...
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    char     EmployeeNo[7];
    double Salary;          // Salary - Used If SalaryNI Is
                            // Positive ( >= 0 )
    short    SalaryNI;      // Salary NULL Indicator - Used
                            // To Determine If Salary
                            // Value Should Be NULL
EXEC SQL END DECLARE SECTION;
...

// Declare A Static Cursor
EXEC SQL DECLARE C1 CURSOR FOR SELECT EMPNO, DOUBLE(SALARY)
    FROM EMPLOYEE;

// Open The Cursor
EXEC SQL OPEN C1;

// If The Cursor Was Opened Successfully, Retrieve And
// Display All Records Available
while (sqlca.sqlcode == SQL_RC_OK)
{
    // Retrieve The Current Record From The Cursor
    EXEC SQL FETCH C1 INTO :EmployeeNo, :Salary :SalaryNI;

    // If The Salary Value For The Record Is NULL, ...
    if (SalaryNI < 0)
    {
        printf("No salary information is available for ");
        printf("employee %s\n", EmployeeNo);
    }
}

// Close The Open Cursor
EXEC SQL CLOSE C1;
...
```

Indicator variables can also be used to send null values to a database when an insert or update operation is performed. When processing `INSERT` and `UPDATE` SQL statements, the DB2 Database Manager examines the value of any indicator variable provided first. If it contains a negative value, the DB2 Database Manager assigns a null value to the appropriate column, provided null values are allowed. (If the indicator variable is set to zero or contains a positive number, or if no indicator variable is used, the DB2 Database Manager assigns the value stored in the corresponding host variable to the appropriate column instead.) Thus, the code used in a C/C++ source code file to assign a null value to a column in a table would look something like:

```
ValueInd = -1;
EXEC SQL INSERT INTO TAB1 VALUES (:Value :ValueInd);
```

# The SQLCA data structure

So far, we've only looked at how host variables and indicator variables are used to move data between embedded SQL applications and database objects. However, there are times when an embedded SQL application needs to communicate with the DB2 Database Manager itself. Two special SQL data structures are used to establish this vital communication link: the *SQL Communications Area* (SQLCA) data structure and the *SQL Descriptor Area* (SQLDA) data structure.

The SQLCA data structure contains a collection of elements that are updated by the DB2 Database Manager each time an SQL statement or a DB2 administrative API function is executed. In order for the DB2 Database Manager to populate this data structure, it must exist. Therefore, any application that contains embedded SQL or that calls one or more administrative APIs must define at least one SQLCA data structure variable. In fact, such an application will not compile successfully if an SQLCA data structure variable does not exist. The following table lists the elements that make up an SQLCA data structure variable.

| Element name | Data type | Description |
|---|---|---|
| `sqlcaid` | `CHAR(8)` | An eye catcher for storage dumps. To help visually identify the data structure, this element normally contains the value `"SQLCA"`. |
| `sqlcabc` | `INTEGER` | The size, in bytes, of the SQLCA data structure itself. This element should always contain the value `136`. |
| `sqlcode` | `INTEGER` | The SQL return code value. A value of `0` indicates successful execution, a positive value indicates successful execution with warnings, and a negative value indicates an error. Refer to the *DB2 UDB Message Reference,* Volumes 1 and 2 product manuals to obtain more information about a specific SQL return code value. |
| `sqlerrml` | `SMALLINT` | The size, in bytes, of the data stored in the `sqlerrmc` element of this structure. This value can be any number between `0` and `70`; a value of `0` indicates that no data has been stored in the `sqlerrmc` field. |
| `sqlerrmc` | `CHAR(70)` | One or more error message tokens, separated by the value `"0xFF"`, that are to be substituted for variables in the descriptions of warning/error conditions. |

| | | |
|---|---|---|
| | | This element is also used when a successful connection is established. |
| sqlerrp | CHAR(8) | A diagnostic value that represents the type of DB2 server currently being used. This value begins with a 3-letter code identifying the product version and release, and is followed by 5 digits that identify the modification level of the product. For example, SQL08014 means DB2 Universal Database, version 8, release 1, modification level 4. If the sqlcode element contains a negative value, this element will contain an 8-character code that identifies the module that reported the error. |
| sqlerrd | INTEGER ARRAY | An array of six integer values that provide additional diagnostic information when an error occurs. (Refer to the next table in this panel for more information about the diagnostic information that can be returned in this element.) |
| sqlwarn | CHAR(11) | An array of character values that serve as warning indicators; each element of the array contains either a blank or the letter W. If compound SQL was used, this field will contain an accumulation of the warning indicators that were set for all substatements executed in the compound SQL statement block. (Refer to the third table in this panel for more information about the types of warning information that can be returned in this element.) |
| sqlstate | CHAR(5) | The SQLSTATE value that identifies the outcome of the most recently executed SQL statement. For more on SQLSTATE values, see SQLSTATEs on page29 . |

Now let's look at the elements of the sqlca.sqlerrd array:

| Array element | Description |
|---|---|
| sqlerrd[0] | If a connection was successfully established, this element will contain the expected difference in length of mixed character data (CHAR data types) when it is converted from the application code page used to the database code page used. A value of 0 or 1 indicates that no expansion is anticipated; a positive value greater than 1 indicates a possible expansion in length; and a negative value indicates a possible reduction in length. |
| sqlerrd[1] | If a connection was successfully established, this element will |

| | |
|---|---|
| | contain the expected difference in length of mixed character data (CHAR data types) when it is converted from the database code page used to the application code page used. A value of `0` or `1` indicates that no expansion is anticipated; a positive value greater than `1` indicates a possible expansion in length; and a negative value indicates a possible reduction in length. If the SQLCA data structure contains information for compound SQL, this element will contain the number of substatements that failed (if any). |
| `sqlerrd[2]` | If the SQLCA data structure contains information for a `CONNECT` SQL statement that executed successfully, this element will contain the value `"1"` if the connected database is updatable and the value `"2"` if the connected database is read-only.<br><br>If the SQLCA data structure contains information for a `PREPARE` SQL statement that executed successfully, this element will contain an estimate of the number of rows that will be returned in a result data set when the prepared statement is executed.<br><br>If the SQLCA data structure contains information for an `INSERT`, `UPDATE`, or `DELETE` SQL statement that executed successfully, this element will contain a count of the number of rows that were affected by the operation.<br><br>If the SQLCA data structure contains information for compound SQL, this element will contain a count of the number of rows that were affected by the substatements in the compound SQL statement block. |
| `sqlerrd[3]` | If the SQLCA data structure contains information for a `CONNECT` SQL statement that executed successfully, this element will contain the value `"0"` if one-phase commit from a down-level client is being used, the value `"1"` if one-phase commit is being used, the value `"2"` if one-phase, read-only commit is being used, and the value `"3"` if two-phase commit is being used.<br><br>If the SQLCA data structure contains information for a `PREPARE` SQL statement that executed successfully, this element will contain a relative cost estimate of the resources needed to prepare the statement specified.<br><br>If the SQLCA data structure contains information for compound SQL, this element will contain a count of the number of substatements in the compound SQL statement block that executed successfully. |
| `sqlerrd[4]` | If the SQLCA data structure contains information for a `CONNECT` SQL statement that executed successfully, this element will contain the value `"0"` if server authentication is being used, the value `"1"` if client authentication is being used, the value `"2"` if authentication is being handled by DB2 Connect, the value `"3"` if authentication is being handled by |

| | |
|---|---|
| | DCE Security Services, and the value "255" if the way authentication is being handled cannot be determined.<br><br>If the SQLCA data structure contains information for anything else, this element will contain a count of the total number of rows that were inserted, updated, or deleted as a result of the DELETE rule of one or more referential integrity constraints or the activation of one or more triggers. (If the SQLCA data structure contains information for compound SQL, this element will contain a count of all such rows for each substatement in the compound SQL statement block that executed successfully.) |
| sqlerrd[5] | For partitioned databases, this element contains the partition number of the partition that encountered an error or warning. If no errors or warnings were encountered, this element will contain the partition number of the partition that serves as the coordinator node. |

And now let's look at the elements of the of the sqlca.sqlwarn array:

| **Array element** | **Description** |
|---|---|
| sqlwarn[0] | This element is blank if all other elements in the array are blank; it contains the character W if one or more of the other elements available is not blank. |
| sqlwarn[1] | This element contains the character W if the value for a column with a character string data type was truncated when it was assigned to a host variable; it contains the character N if the null-terminator for the string was truncated. |
| sqlwarn[2] | This element contains the character W if null values were eliminated from the arguments passed to a function. |
| sqlwarn[3] | This element contains the character W if the number of values retrieved does not equal the number of host variables provided. |
| sqlwarn[4] | This element contains the character W if an UPDATE or DELETE SQL statement that does not contain a WHERE clause was prepared. |
| sqlwarn[5] | This element is reserved for future use. |
| sqlwarn[6] | This element contains the character W if the result of a date calculation was adjusted to avoid an invalid date value. |
| sqlwarn[7] | This element is reserved for future use. |
| sqlwarn[8] | This element contains the character W if a character that could not be converted was replaced with a substitution character. |
| sqlwarn[9] | This element contains the character W if one or more errors in an arithmetic expression were ignored during column function processing. |
| sqlwarn[10] | This element contains the character W if a conversion error |

| | |
|---|---|
| | occurred while converting a character data value in another element of the SQLCA data structure variable. |

## The SQLDA data structure

The SQL Descriptor Area (SQLDA) data structure contains a collection of elements that are used to provide detailed information to the PREPARE, OPEN, FETCH, and EXECUTE SQL statements. This data structure consists of a header followed by an array of structures, each of which describes a single host variable or a single column in a result data set. The following table lists the elements that make up an SQLDA data structure variable.

| Element name | Data type | Description |
|---|---|---|
| sqldaid | CHAR(8) | An eye catcher for storage dumps. To help visually identify the data structure, this element normally contains the value "SQLDA ". |
| sqldabc | INTEGER | The size, in bytes, of the SQLDA data structure itself. The value assigned to this element is determined using the equation sqldabc = 16 + (44 * sqln). |
| sqln | SMALLINT | The total number of elements in the sqlvar array. |
| sqld | SMALLINT | This element can indicate one of two things: either the number of columns in the result data set returned by a DESCRIBE or a PREPARE SQL statement, or the number of host variables described by the elements in the sqlvar array. |
| sqlvar | STRUCTURE ARRAY | An array of data structures that contain information about host variables or result data set columns. |

In addition to this basic information, an SQLDA data structure variable contains an arbitrary number of occurrences of sqlvar data structures (which are referred to as *SQLVAR variables*). The information stored in each SQLVAR variable depends on the location where the corresponding SQLDA data structure variable is used: When used with a PREPARE or a DESCRIBE SQL statement, each SQLVAR variable will contain information about a column that will exist in the result data set produced when the prepared SQL statement is executed. (If any of the columns have a large object (LOB) or user-defined data type, the number of SQLVAR variables used will be doubled and the seventh byte of the character string value stored in the sqldaid element of the SQLDA data structure variable will be assigned the value "2".) On the other hand, when the SQLDA data structure variable is used with an OPEN, FETCH, or EXECUTE SQL statement, each SQLVAR variable will contain information about

a host variable whose value is to be passed to the DB2 Database Manager.

Two types of SQLVAR variables are used: *base SQLVARs* and *secondary SQLVARs.* Base SQLVARs contain basic information (such as data type code, length attribute, column name, host variable address, and indicator variable address) for result data set columns or host variables. The elements that make up a base SQLVAR data structure variable are shown in the following table.

| Element name | Data type | Description |
|---|---|---|
| `sqltype` | `SMALLINT` | The data type of a host variable used, or the data type of a column in the result data set produced. |
| `sqllen` | `SMALLINT` | The size (length), in bytes, of a host variable used, or the size of a column in the result data set produced. |
| `sqldata` | `Pointer` | A pointer to a location in memory where the data for a host variable used is stored, or a pointer to a location in memory where data for a column in the result data set produced is to be stored. |
| `sqlind` | `Pointer` | A pointer to a location in memory where the data for the null indicator variable associated with a host variable used is stored, or a pointer to a location in memory where the data for the null indicator variable associated with a column in the result data set produced is to be stored. |
| `sqlname` | `VARCHAR(30)` | The unqualified name of a host variable or a column in the result data set produced. |

On the other hand, secondary SQLVARs contain either the distinct data type name for distinct data types or the length attribute of the column or host variable and a pointer to the buffer that contains the actual length of the data for LOB data types. Secondary SQLVAR entries are only present if the number of SQLVAR entries is doubled because LOBs or distinct data types are used: If locators or file reference variables are used to represent LOB data types, secondary SQLVAR entries are not used.

The information stored in an SQLDA data structure variable, along with the information stored in any corresponding SQLVAR variables, may be placed there manually (using the appropriate programming language statements), or can be generated automatically by executing the `DESCRIBE` SQL statement.

Both an SQLCA data structure variable and an SQLDA data structure variable can be created by embedding the appropriate form of the `INCLUDE` SQL statement (`INCLUDE SQLCA` and `INCLUDE SQLDA`, respectively) within an embedded SQL source code file.

# Establishing a database connection

In order to perform any type of operation against a database, you must first establish a connection to that database. With embedded SQL applications, database connections are made (and in some cases terminated) by executing the `CONNECT` SQL statement. (The `RESET` option of the `CONNECT` statement is used to terminate a connection.) During the connection process, the information needed to establish a connection -- such as an authorization ID and a corresponding password of an authorized user -- is passed to the database specified for validation. Often, this information is collected at application run time and passed to the `CONNECT` statement by way of one or more host variables.

Embedded SQL applications can use two different types of connection semantics. These two types, known simply as *Type 1* and *Type 2*, support two different types of transaction behavior: Type 1 connections support only one database connection per transaction (referred to as a *remote unit of work*) while Type 2 connections support any number of database connections per transaction (referred to as an *application-directed distributed unit of work*). Essentially, when Type 1 connections are used, an application can only be connected to one database at a time. Once a connection to a database is established and a transaction is started, that transaction must either be committed or rolled back before another database connection can be established. On the other hand, when Type 2 connections are used, an application can be connected to several different databases at the same time, and each database connection will have its own set of transactions. (The actual type of connection semantics an application will use is determined by the value assigned to the `CONNECT`, `SQLRULES`, `DISCONNECT`, and `SYNCPOINT` SQL precompiler options when the application is precompiled.)

---

# Preparing and executing SQL statements

When static SQL statements are embedded in an application, they are executed as they are encountered. However, when dynamic SQL statements are used, they can be processed in two ways:

- *Prepare and execute:* This approach separates the preparation of the SQL statement from its actual execution and is typically used when an SQL statement is to be executed repeatedly. This method is also used when an application needs advance information about the columns that will exist in the result data set produced when a `SELECT` SQL statement is executed. The SQL statements `PREPARE` and `EXECUTE` are used to process dynamic SQL statements in this manner.
- *Execute immediately:* This approach combines the preparation and the execution of an SQL statement into a single step and is typically used when an SQL statement is to be executed only once. This method is also used when the application does not need additional information about the result

data set that will be produced, if any, when the SQL statement is executed. The SQL statement `EXECUTE IMMEDIATE` is used to process dynamic SQL statements in this manner.

Dynamic SQL statements that are prepared and executed (using either method) at run time are not allowed to contain references to host variables. They can, however, contain parameter markers in place of constants and/or expressions. Parameter markers are represented by the question mark (?) character. They indicate where in the SQL statement the current value of one or more host variables or elements of an SQLDA data structure variable are to be substituted when the statement is executed. (Parameter markers are typically used where a host variable would be referenced if the SQL statement being executed were static.) Two types of parameter markers are available: *typed* and *untyped.*

A typed parameter marker is one that is specified along with its target data type. Typed parameter markers have this general form:

```
CAST(? AS DataType)
```

This notation does not imply that a function is called, but rather it promises that the data type of the value replacing the parameter marker at application run time will either be the data type specified or a data type that can be converted to the data type specified. For example, consider the following SQL statement:

```
UPDATE EMPLOYEE SET LASTNAME = CAST(? AS VARCHAR(12))
WHERE EMPNO = '000050'
```

Here, the value for the LASTNAME column is provided at application run time, and the data type of that value will be either `VARCHAR(12)` or a data type that can be converted to `VARCHAR(12)`.

An untyped parameter marker, on the other hand, is specified without a target data type and has the form of a single question mark (?). The data type of an untyped parameter marker is determined by the context in which it is used. For example, in the following SQL statement, the value for the LASTNAME column is provided at application run time, and it is assumed that the data type of that value will be compatible with the data type that has been assigned to the LASTNAME column of the EMPLOYEE table.

```
UPDATE EMPLOYEE SET LASTNAME = ? WHERE EMPNO = '000050'
```

When parameter markers are used in embedded SQL applications, values that are to be substituted for parameter markers placed in an SQL statement must be provided as additional parameters to the `EXECUTE` or the `EXECUTE IMMEDIATE` SQL statement when either is used to execute the SQL statement specified. The following example, written in the C programming language, illustrates how actual values would be provided for parameter markers that have been coded in a simple `UPDATE` SQL statement:

```
...
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    char     SQLStmt[80];
    char     JobType[10];
EXEC SQL END DECLARE SECTION;
...

// Define A Dynamic UPDATE SQL Statement That Uses A
// Parameter Marker
strcpy(SQLStmt, "UPDATE EMPLOYEE SET JOB = ? ");
strcat(SQLStmt, "WHERE JOB = 'DESIGNER'");

// Populate The Host Variable That Will Be Used In
// Place Of The Parameter Marker
strcpy(JobType, "MANAGER");

// Prepare The SQL Statement
EXEC SQL PREPARE SQL_STMT FROM :SQLStmt;

// Execute The SQL Statement
EXEC SQL EXECUTE SQL_STMT USING :JobType;
...
```

---

# Retrieving and processing results

Regardless of whether a static SQL statement or a dynamic SQL statement is used in an embedded SQL application, once the statement has been executed, any results produced will need to be retrieved and processed. If the SQL statement was anything other than a SELECT or a VALUES statement, the only additional processing required after execution is a check of the SQLCA data structure variable to ensure that the statement executed as expected. However, if a SELECT statement or VALUES statement was executed, additional steps are needed to retrieve each row of data from the result data set produced.

When a SELECT statement or a VALUES statement is executed from within an application, DB2 UDB uses a mechanism known as a *cursor* to retrieve data values from any result data set produced. The name *cursor* probably originated from the blinking cursor found on early computer screens, and just as that cursor indicated the current position on the screen and identified where typed words would appear next, a DB2 UDB cursor indicates the current position in the result data set (i.e., the current row) and identifies the row of data that will be returned to the application next. The following steps must be performed in order if a cursor is to be incorporated into an embedded SQL application:

1. Declare (define) a cursor along with its type (read-only or updatable), and associate it with the desired query (SELECT or VALUES SQL statement). This is done by executing the DECLARE CURSOR statement.
2. Open the cursor. This will cause the corresponding query to be executed and a result data set to be produced. This is done by executing the OPEN statement.

3. Retrieve (fetch) each row in the result data set, one by one, until an end-of-data condition occurs. Each time a row is retrieved from the result data set, the cursor is automatically moved to the next row. This is done by repeatedly executing the `FETCH` statement; host variables or an SQLDA data structure variable are used in conjunction with a `FETCH` statement to extract a row of data from a result data set.
4. If appropriate, modify or delete the current row, but only if the cursor is an updatable cursor. This is done by executing the `UPDATE` statement or the `DELETE` statement.
5. Close the cursor. This action will cause the result data set that was produced when the corresponding query was executed to be deleted. This is done by executing the `CLOSE` statement.

Now that we have seen the steps that must be performed in order to use a cursor, let's examine how these steps are coded in an application. The following example, written in the C programming language, illustrates how a cursor would be used to retrieve the results of a `SELECT` SQL statement:

```
...
// Declare The SQL Host Memory Variables
EXEC SQL BEGIN DECLARE SECTION;
    char      EmployeeNo[7];
    char      LastName[16];
EXEC SQL END DECLARE SECTION;
...

// Declare A Cursor
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT EMPNO, LASTNAME
    FROM EMPLOYEE
    WHERE JOB = 'DESIGNER';

// Open The Cursor
EXEC SQL OPEN C1;

// Fetch The Records
while (sqlca.sqlcode == SQL_RC_OK)
{
    // Retrieve A Record
    EXEC SQL FETCH C1
    INTO :EmployeeNo, :LastName;

    // Process The Information Retrieved
    if (sqlca.sqlcode == SQL_RC_OK)
        ...
}

// Close The Cursor
EXEC SQL CLOSE C1;
...
```

If you know in advance that only one row of data will be produced in response to a query, there are two other ways to copy the contents of that row to host variables within an application program, by executing either the `SELECT INTO` statement or the `VALUES INTO` statement. Like the `SELECT` SQL statement,

the `SELECT INTO` statement can be used to construct complex queries. However, unlike the `SELECT` statement, the `SELECT INTO` statement requires a list of valid host variables to be supplied as part of its syntax; it cannot be used dynamically. Additionally, if the result data set produced when the `SELECT INTO` statement is executed contains more than one record, the operation will fail and an error will be generated. (If the result data set produced is empty, a `NOT FOUND` warning will be generated.)

Like the `SELECT INTO` statement, the `VALUES INTO` statement can be used to retrieve the data associated with a single record and copy it to one or more host variables. And, like the `SELECT INTO` statement, when the `VALUES INTO` statement is executed, all data retrieved is stored in a result data set. If this result data set contains only one record, the first value in that record is copied to the first host variable specified, the second value is copied to the second host variable specified, and so on. However, the `VALUES INTO` statement cannot be used to construct complex queries in the same way that the `SELECT INTO` statement can.

Again, if the result data set produced when the `VALUES INTO` statement is executed contains more than one record, the operation will fail and an error will be generated. (If the result data set produced is empty, a `NOT FOUND` warning will be generated.)

---

# Managing transactions

A *transaction* (also known as a *unit of work*) is a sequence of one or more SQL operations grouped together as a single unit, usually within an application process. Such a unit is called `atomic` because it is indivisible -- either all of its work is carried out or none of its work is carried out. A given transaction can be comprised of any number of SQL operations, from a single operation to many hundreds or even thousands, depending upon what is considered a single step within your business logic.

The initiation and termination of a single transaction define points of data consistency within a database: Either the effects of all operations performed within a transaction are applied to the database and made permanent (committed), or the effects of all operations performed are backed out (rolled back) and the database is returned to the state it was in before the transaction was initiated. In most cases, transactions are initiated the first time an executable SQL statement is executed after a connection to a database has been established, or immediately after a pre-existing transaction has been terminated. Once initiated, transactions can be implicitly terminated using the *automatic commit* feature. Using this feature, each executable SQL statement is treated as a single transaction. If the statement executes successfully, any changes made by that statement are applied to the database. If the statement fails, any changes are discarded. Transactions can also be explicitly terminated by executing either the `COMMIT` or the `ROLLBACK` SQL statement. In either case, all transactions associated with a particular database should be completed before the connection to that database is terminated.

# Putting it all together

Now that we have examined some of the basic components used to construct embedded SQL applications, let's see how each is typically coded in an embedded SQL application. A simple embedded SQL application, written in the C programming language using static SQL, that obtains and prints employee identification numbers, last names, and salaries for all employees who have the job title "designer" might look something like:

```
#include <stdio.h>
#include <string.h>
#include <sql.h>

int main()
{
    // Include The SQLCA Data Structure Variable
    EXEC SQL INCLUDE SQLCA;

    // Define The SQL Host Variables Needed
    EXEC SQL BEGIN DECLARE SECTION;
        char     EmployeeNo[7];
        char     LastName[16];
        double   Salary;
        short    SalaryNI;
    EXEC SQL END DECLARE SECTION;

    // Connect To The Appropriate Database
    EXEC SQL CONNECT TO SAMPLE USER db2admin USING ibmdb2;

    // Declare A Static Cursor
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT EMPNO,
               LASTNAME,
               DOUBLE(SALARY)
        FROM EMPLOYEE
        WHERE JOB = 'DESIGNER';

    // Open The Cursor
    EXEC SQL OPEN C1;

    // If The Cursor Was Opened Successfully, Retrieve And
    // Display All Records Available
    while (sqlca.sqlcode == SQL_RC_OK)
    {
        // Retrieve The Current Record From The Cursor
        EXEC SQL FETCH C1
            INTO :EmployeeNo,
                 :LastName,
                 :Salary :SalaryNI;

        // Display The Record Retrieved
        if (sqlca.sqlcode == SQL_RC_OK)
        {
                printf("%-8s %-16s ", EmployeeNo,
                    LastName);
                if (SalaryNI >= 0)
```

```
                    printf("%lf\n", Salary);
                else
                    printf("Unknown\n");
            }
        }

        // Close The Open Cursor
        EXEC SQL CLOSE C1;

        // Commit The Transaction
        EXEC SQL COMMIT;

        // Terminate The Database Connection
        EXEC SQL DISCONNECT CURRENT;

        // Return Control To The Operating System
        return(0);
    }
```

On the other hand, a simple embedded SQL application, written in the C programming language using dynamic SQL, that changes all job titles "designer" to "manager" might look something like:

```
#include <stdio.h>
#include <string.h>
#include <sql.h>

int main()
{
    // Include The SQLCA Data Structure Variable
    EXEC SQL INCLUDE SQLCA;

    // Define The SQL Host Variables Needed
    EXEC SQL BEGIN DECLARE SECTION;
        char     SQLStmt[80];
        char     JobType[10];
    EXEC SQL END DECLARE SECTION;

    // Connect To The Appropriate Database
    EXEC SQL CONNECT TO SAMPLE USER db2admin USING ibmdb2;

    // Define A Dynamic UPDATE SQL Statement That Uses A
    // Parameter Marker
    strcpy(SQLStmt, "UPDATE EMPLOYEE SET JOB = ? ");
    strcat(SQLStmt, "WHERE JOB = 'DESIGNER'");

    // Populate The Host Variable That Will Be Used In
    // Place Of The Parameter Marker
    strcpy(JobType, "MANAGER");

    // Prepare The SQL Statement
    EXEC SQL PREPARE SQL_STMT FROM :SQLStmt;

    // Execute The SQL Statement
    EXEC SQL EXECUTE SQL_STMT USING :JobType;

    // Commit The Transaction
    EXEC SQL COMMIT;
```

```
    // Terminate The Database Connection
    EXEC SQL DISCONNECT CURRENT;

    // Return Control To The Operating System
    return(0);
}
```

# Section 4. Diagnostics and error handling

## Using the **WHENEVER** statement

In The SQLCA data structure on page 11, we saw that the SQL Communications Area (SQLCA) data structure contains a collection of elements that are updated by the DB2 Database Manager each time an SQL statement is executed. One element of that structure, the `sqlcode` element, is assigned a value that indicates the success or failure of the SQL statement executed. (A value of `0` indicates successful execution, a positive value indicates successful execution with warnings, and a negative value indicates that an error occurred.) At a minimum, an embedded SQL application should always check the `sqlcode` value produced (often referred to as the *SQL return code*) immediately after an SQL statement is executed. Whenever an SQL statement fails to execute as expected, users should be notified that an error or warning condition occurred. Whenever possible, they should be provided with sufficient diagnostic information so they can locate and correct the problem.

As you might imagine, checking the SQL return code after each SQL statement is executed can add additional overhead to an application, especially when an application contains a large number of SQL statements. However, because every SQL statement coded in an embedded SQL application source code file must be processed by the SQL precompiler, it is possible to have the precompiler automatically generate the source code needed to check SQL return codes. This is accomplished by embedding one or more forms of the `WHENEVER` SQL statement into a source code file.

The `WHENEVER` statement tells the precompiler to generate source code that evaluates SQL return codes and branches to a specified label whenever an error, warning, or out-of-data condition occurs. (If the `WHENEVER` statement is not used, the default behavior is to ignore SQL return codes and continue processing as if no problems have been encountered.) Four forms of the `WHENEVER` statement are available, one for each of the three different types of error/warning conditions for which the `WHENEVER` statement can be used to check, and one to turn error checking off:

- `WHENEVER SQLERROR GOTO [`*`Label`*`]`: Instructs the precompiler to generate source code that evaluates SQL return codes and branches to the label specified whenever a negative `sqlcode` value is generated.
- `WHENEVER SQLWARNING GOTO [`*`Label`*`]`: Instructs the precompiler to generate source code that evaluates SQL return codes and branches to the label specified whenever a positive `sqlcode` value (other than the value `"100"`) is generated.
- `WHENEVER NOT FOUND GOTO [`*`Label`*`]`: Instructs the precompiler to generate source code that evaluates SQL return codes and branches to the label specified whenever an `sqlcode` value of `100` or an `sqlstate` value of `02000` is generated.
- `WHENEVER [SQLERROR | SQL WARNING | NOT FOUND] CONTINUE`: Instructs the precompiler to ignore the SQL return code and continue with

the next instruction in the application.

A source code file can contain any combination of these four forms of the WHENEVER statement, and the order in which the first three forms appear is insignificant. However, once any form of the WHENEVER statement is used, the SQL return codes of all subsequent SQL statements executed will be evaluated and processed accordingly until the application ends or until another WHENEVER statement alters this behavior.

The following example, written in the C programming language, illustrates how the WHENEVER statement could typically be used to trap and process out-of-data errors:

```
...
// Include The SQLCA Data Structure Variable
EXEC SQL INCLUDE SQLCA;

// Set Up Error Handler
EXEC SQL WHENEVER NOT FOUND GOTO NOT_FOUND_HANDLER;

// Connect To The Appropriate Database
EXEC SQL CONNECT TO SAMPLE USER db2admin USING ibmdb2;

// Execute A SELECT INTO SQL Statement (If A "DATA NOT
// FOUND" Situation Occurs, The Code Will Branch To
// The NOT_FOUND_HANDLER Label)
EXEC SQL SELECT EMPNO INTO :EmployeeNo
    FROM RSANDERS.EMPLOYEE
    WHERE JOB = 'CODER';
...

// Disable All Error Handling
EXEC SQL WHENEVER NOT FOUND CONTINUE;

// Prepare To Return To The Operating System
goto EXIT;

// Define A Generic "Data Not Found" Handler
NOT_FOUND_HANDLER:
    printf("NOT FOUND: SQL Code = %d\n", sqlca.sqlcode);
    EXEC SQL ROLLBACK;
    goto EXIT;

EXIT:

// Terminate The Database Connection
EXEC SQL DISCONNECT CURRENT;

// Return Control To The Operating System
return(0);
```

Unfortunately, the code that is generated when the WHENEVER SQL statement is used relies on GO TO branching instead of call/return interfaces to transfer control to the appropriate error handling section of an embedded SQL application. As a result, when control is passed to the source code that is used to process errors and warnings, the application has no way of knowing where

control came from, nor does it have any way of knowing where it should return control to after the error or warning has been properly handled. For this reason, about the only thing an application can do when control is passed to a `WHENEVER` statement error handling label is to display the error code generated, roll back the current transaction, and return control to the operating system.

---

## The Get Error Message API

Among other things, most editions of DB2 UDB and the DB2 Application Development Client contain a rich set of functions that are referred to as the *administrative APIs* (Application Programming Interfaces). These APIs are designed to provide services other than the data storage, manipulation, and retrieval functionality that SQL provides to DB2 UDB applications. (Essentially, any database operation that can be performed from the Command Line Processor by executing a DB2 command can be performed from within an application by calling an administrative API.)

Earlier, we saw that the SQL Communications Area (SQLCA) data structure contains a collection of elements that are updated by the DB2 Database Manager each time an SQL statement is executed and that one element of that structure, the `sqlcode` element, is assigned a value that indicates the success or failure of the SQL statement executed. The value that gets assigned to the `sqlcode` element is actually a coded number. A special administrative API can be used to translate the coded number into a meaningful description that can then be displayed to the user. This API is known as the *Get Error Message* API. The basic syntax used to call it from a high-level programming source code file is as follows for C/C++ applications:

```
sqlaintp (char          *pBuffer,
          short         sBufferSize,
          short         sLineWidth,
          struct sqlca  *pSQLCA);
```

And here's the syntax for other high-level programming language applications:

```
sqlgintp (short         sBufferSize,
          short         sLineWidth,
          struct sqlca  *pSQLCA,
          char          *pBuffer);
```

Let's look at the components of the syntax in more detail:

- `pBuffer`: Identifies a location in memory where the Get Error Message API is to store any message text retrieved.
- `sBufferSize`: Identifies the size, in bytes, of the memory storage buffer to which any message text retrieved should be written.
- `sLineWidth`: Identifies the maximum number of characters that one line of

message text should contain before a line break is inserted. A value of `0` indicates that the message text is to be returned without line breaks.

- pSQLCA: Identifies a location in memory where an SQL Communications Area (SQLCA) data structure variable is stored.

Each time the Get Error Message API is called, the value stored in the `sqlcode` element of the SQLCA data structure variable provided is used to locate and retrieve appropriate error message text from a message file that is shipped with DB2 UDB. The following example, written in the C programming language, illustrates how the Get Error Message API would typically be used to obtain and display the message associated with any SQL return code generated:

```
...
// Include The SQLCA Data Structure Variable
EXEC SQL INCLUDE SQLCA;

// Declare The Local Memory Variables
long  RetCode = SQL_RC_OK;
char  ErrorMsg[1024];
...

// Perform Some SQL Operation
...

// If An Error Occurred, Obtain And Display
// Any Diagnostic Information Available
if (sqlca.sqlcode != SQL_RC_OK)
{
    // Retrieve The Error Message Text For The Error
    // Code Generated
    RetCode = sqlaintp(ErrorMsg, sizeof(ErrorMsg), 70, &sqlca);
    switch (RetCode)
    {
    case -1:
        printf("ERROR : Insufficient memory.\n");
        break;
    case -3:
        printf("ERROR : Message file is inaccessible.\n");
        break;
    case -5:
        printf("ERROR : Invalid SQLCA, bad buffer, ");
        printf("or bad buffer length specified.\n");
        break;
    default:
        printf("%s\n", ErrorMsg);
        break;
    }
}
...
```

As you can see in this example, when the Get Error Message API is called, it returns a value that indicates whether or not it executed successfully. In this case, the return code produced is checked. If an error did occur, a message is returned to the user explaining why the API failed. If the API was successful, the message retrieved is returned to the user instead.

# SQLSTATEs

DB2 UDB (as well as other relational database products) uses a set of error message codes known as *SQLSTATEs* to provide supplementary diagnostic information for warnings and errors. SQLSTATEs are alphanumeric strings that are five characters (bytes) in length and have the format *ccsss*, where *cc* indicates the error message class and *sss* indicates the error message subclass. Like SQL return code values, SQLSTATE values are written to an element (the `sqlstate` element) of an SQLCA data structure variable used each time an SQL statement is executed. And just as the Get Error Message API can be used to convert any SQL return code value generated into a meaningful description, another API -- the *Get SQLSTATE Message* API -- can be used to convert an SQLSTATE value into a meaningful description as well. By including either (or both) of these APIs in your embedded SQL applications, you can always return meaningful information to the end user whenever error and/or warning conditions occur.

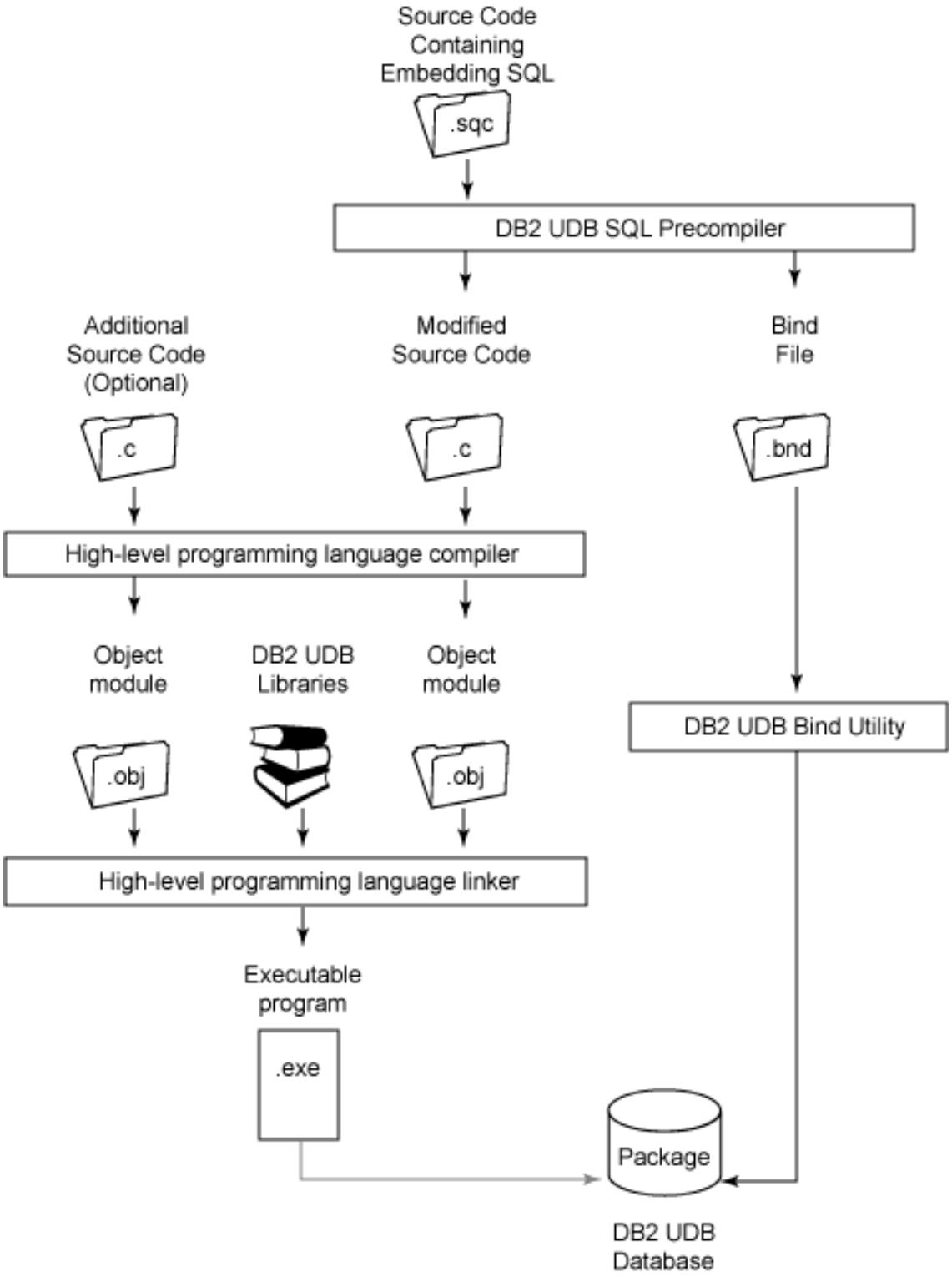
# Section 5. Creating executable applications

## The basic process

So far, we have looked at some of the basic steps of embedding SQL statements in high-level programming language source code files, but we have only hinted at how source code files containing embedded SQL statements are converted into a working program. Once a source code file has been written, the following steps must be performed, in the order shown, before an application that interacts with a DB2 UDB database will be created:

1. All source code file(s) containing embedded SQL statements must be precompiled to convert the embedded SQL statements used into DB2-specific function calls and to create a corresponding package. You must be connected to a database in order to run the SQL precompiler. All packages created can be stored in the database being used by the SQL precompiler, or they can be written to a special file known as a *bind file,* which can then be bound to any valid DB2 UDB database later.
2. All high-level programming language source code files produced by the SQL precompiler (and any additional source code files needed) must be compiled to create object modules.
3. All appropriate object modules must be linked with high-level programming language libraries and the DB2 UDB libraries to create an executable program.
4. If the packages for the files that were processed by the SQL precompiler have not already been bound to the appropriate database, they must be bound using the bind files produced by the SQL precompiler.

The following illustration outlines the basic embedded SQL source code file-to-executable application conversion process when deferred binding is used. (We'll discuss deferred binding in more detail in Creating and binding packages on page32 .)




## Precompiling, compiling, and linking

During the precompile process, a source code file containing embedded SQL statements is converted into a source code file that is made up entirely of high-level programming language statements. (The embedded SQL statements

themselves are commented out and DB2-specific function calls are stored in their place.) At the same time, a corresponding package that contains (among other things) the access plans that are to be used to process each static SQL statement embedded in the source code file is also produced. (Access plans contain optimized information that the DB2 Database Manager uses to execute SQL statements. Access plans for static SQL statements are produced at precompile time, while access plans for dynamic SQL statements are produced at application run time.) Packages produced by the SQL precompiler can be stored in the database being used by the precompiler as they are generated, or they can be written to an external bind file and bound to any valid DB2 UDB database later (the process of storing this package in the appropriate database is known as *binding*). By default, packages are automatically bound to the database used for precompiling during the precompile process. Unless otherwise specified, the SQL precompiler is also responsible for verifying that all database objects (such as tables and columns) that have been referenced in static SQL statements actually exist, and that all application data types used are compatible with their database counterparts (that's why you need a database connection in order to use the SQL precompiler.)

Once a source code file containing embedded SQL statements has been processed by the SQL precompiler, the high-level programming language source code file that is produced -- and any other source code files used -- must be compiled by a high-level programming language compiler. This compiler is responsible for converting source code files into object modules that the linker can use to create an executable program.

When all of the source code files needed to build an application have been compiled successfully, the resulting object module can be provided as input to the linker. The linker combines object modules, high-level programming language libraries, and DB2 UDB libraries to produce an executable application. In most cases, this executable application exists as an executable file. However, it can also exist as a shared library or a dynamic-link library (DLL) that is loaded and executed by other executable applications

---

# Creating and binding packages

Earlier, we saw that when a source code file containing embedded SQL statements is processed by the SQL precompiler, a package containing data access plans is produced, along with a source code file that is made up entirely of high-level programming language statements. This package must reside in an appropriate DB2 UDB database (i.e., a database that contains data objects that are referenced by the package) before the corresponding application can be executed against that database.

The process of storing such a package in a DB2 UDB database is known as *binding*. By default, packages are automatically bound to the database being used by the SQL precompiler during the precompile process. However, by specifying the appropriate precompiler options, you can elect to store the steps needed to create the package in a separate file (rather than in a database) and

complete the binding process at a later point in time, using a tool known as the *DB2 Binder* (or simply the Binder). This is referred to as *deferred binding*, and is preferable if you want to:

- Defer binding until you have an application program that compiles and links successfully.
- Create a package under a different schema or under multiple schemas.
- Run an application against a database using different options (isolation level, Explain on/off, etc.). By deferring the bind process, you can dynamically change things like the isolation level used without having to rebuild the application.
- Run an application against several different databases. By deferring the bind process, you can build your program once and bind it to any number of appropriate databases. Otherwise, you will have to rebuild the entire application each time you want to run it against a new database.
- Run an application against a database that has been duplicated on several different machines. By deferring the bind process, you can dynamically create your application database on each machine, and then bind your program to the newly created database (possibly as part of your application's installation process).

# Section 6. Conclusion

## Summary

This tutorial introduced you to embedded SQL programming and walked you through the basic steps used to construct an embedded SQL application. At this point, you should know the difference between static SQL and dynamic SQL, and you should know how both types of SQL statements can be embedded in a high-level programming language source code file.

You should know how to declare and use host and indicator variables to move data between an application and a database, and you should be able to analyze the contents of an SQLCA data structure variable to determine whether an embedded SQL statement executed as expected. Furthermore, you should know how to establish a database connection, how to retrieve and process any results produced, and how to terminate transactions.

Finally, you should be familiar with the steps used to convert a source code file containing embedded SQL statements into an executable application.

---

## Resources

For more information on DB2 Universal Database application development:

- *DB2 Version 8 Administration Guide: Implementation*, International Business Machines Corporation, 2002.
- *DB2 Version 8 Application Development Guide: Programming Client Applications*, International Business Machines Corporation, 2002.
- *DB2 Version 8 Application Development Guide: Programming Server Applications*, International Business Machines Corporation, 2002.
- *DB2 Version 8 Application Development Guide: Building and Running Applications*, International Business Machines Corporation, 2002.
- *DB2 Version 8 SQL Reference Guide, Volume 1*, International Business Machines Corporation, 2002.
- *DB2 Version 8 SQL Reference Guide, Volume 2*, International Business Machines Corporation, 2002.

For more information on the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703):

- *DB2 Universal Database v8.1 Certification Exam 703 Study Guide*, Sanders, Roger E., International Business Machines Corporation, 2004.
- *DB2 Universal Database v8 Application Development Certification Guide*, Martineau, David and others, International Business Machines Corporation, 2003.

- *IBM DB2 Information Management -- Training and certification* (http://www.ibm.com/software/data/education//) for information on classes, certifications available and additional resources.

As mentioned earlier, this tutorial is just one tutorial in a series of seven to help you prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The complete list of all tutorials in this series is provided below:

1. *Database objects and Programming Methods*
2. *Data Manipulation*
3. Embedded SQL Programming
4. *ODBC/CLI Programming*
5. *Java Programming*
6. *Advanced Programming*
7. User-Defined Routines

Before you take the certification exam (DB2 UDB V8.1 Application Development, Exam 703) for which this tutorial was created to help you prepare, you should have already taken and passed the DB2 V8.1 Family Fundamentals certification exam (Exam 700). Use the *DB2 V8.1 Family Fundamentals certification prep tutorial series* to prepare for that exam. A set of six tutorials covers the following topics:

- DB2 planning
- DB2 security
- Accessing DB2 UDB data
- Working with DB2 UDB data
- Working with DB2 UDB objects
- Data concurrency

Use the *DB2 V8.1 Database Administration certification prep tutorial series* to prepare for the DB2 UDB V8.1 for Linux, UNIX and Windows Database Administration certification exam (Exam 701). A set of six tutorials covers the following topics:

- Server management
- Data placement
- Database access
- Monitoring DB2 activity
- DB2 utilities
- Backup and recovery

Check out *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus, Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.

# Feedback

## Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .